

1. 本周工作

1. 阅读《C++ concurrency in action》Chapter 7. Designing lock-free concurrent data structures前半部分内容（内容提炼见附录）；

2. 下周工作

1. 了解高清拼接屏项目：部署、SDK等
2. Chapter 7. Designing lock-free concurrent data structures剩下部分

附录

Chapter 7. Designing lock-free concurrent data structures

本章主要内容有：

- 不使用锁，实现为并发设计的数据结构
- 用无锁数据结构管理内存
- 编写无锁数据的指导方针

7.1 Definitions and consequences

有锁编程可能引发的问题：

- 死锁
- 并发性下降

阻塞与非阻塞：

- 使用mutex, condition variable或future来实现同步的数据结构和算法是**阻塞的 (blocking)**
- 不使用阻塞库函数的数据结构和算法是**非阻塞的 (unblocking)**，但并非所有这些数据结构都是**无锁的 (lock-free)**。

7.1.1 Types of nonblocking data structures

重现了第五章（acquire-release模型）中，用atomic_flag实现的自旋锁。

该自旋锁是非阻塞性的，但不是无锁的。

7.1.2 Lock-free data structures

具有资格被称为无锁的数据结构，应支持同时多个线程对它并发访问。

使用compare/exchange loop的数据结构，若在别的线程都被挂起的情况下，可以成功完成，这种数据结构仍是无锁的；若不能完成，则你本质上拥有的是自旋锁——非阻塞但不是无锁的。

7.1.3 Wait-free data structures

确保每个线程都可以在有限步骤内完成它所有操作（不会无限时等待）的数据结构，是wait-free。

设计lock-free或wait-free数据结构，一方面需要保证在有限步骤内完成所有操作，另一方面要保证一个线程内的操作不会导致另一个线程中的操作失败，这非常复杂。因此需要一个好的理由，才开始编码这样的数据结果，确保获得的受益比代价更高。

7.1.4 The pros and cons of lock-free data structures

无锁数据结构的优点：

- 可获得最大的并发性
- 鲁棒性高
 - 在有锁编程中，一个线程在死亡时若未释放它所持有的锁，可能导致整个数据结构被破坏
 - 在无锁编程中，一个线程的死亡只会导致它自身数据的丢失
 - 事实上，我不认同书中上述的观点，在有锁编程中采用lock_guard, 在线程死亡时是可以释放锁的。同时，我觉得抛开代码光谈编程模型，很难比较出鲁棒性的高低。
- 没有死锁

无锁数据结构的缺点：

- 引入活锁（live lock）
 - **定义**：当两个线程同时访问一个数据结构，每个线程都由于对方线程发生的改变，而重新开始（restart）自己的工作
 - 可以想象两个人都像通过一座独木桥，每人往桥上走一步，发现对面有人，便后退。这种前进后退、前进后退的反复操作。
 - **性能**：活锁一般是短期存在的，所以它会影响性能，但不会造成程序无法继续运行
 - **免等待（wait-free）**：在有限步骤内能完成操作的数据结构被称为免等待的，它不受无锁的影响
 - *By definition, wait-free code can't suffer from live lock*
 - 实现wait-free会更加复杂；同时，对于只有一个线程访问数据结构时，相比non-wait-free的数据结构，它需要多执行几步
- 原子操作比非原子操作要慢得多，未必会带来性能提升

真正的性能优化（使用有锁还是无锁）是经过测试检验获得的。

7.2 Examples of lock-free data structures

数据结构中，只有atomic_flag保证不使用锁。

“

Only std::atomic_flag is guaranteed not to use locks.

7.2.1 Writing a thread-safe stack without locks

添加节点的步骤：

1. 创建新节点
2. 将新节点的next指向stack的head
3. 将stack的head赋值成新节点

当两个线程同时添加节点时，竞态可能在步骤2, 3中产生。

弹出并获得节点的步骤：

1. 读取当前head
2. 读取head->next
3. 将head->next赋值给head
4. 返回从节点中获得的data
5. 删除要弹出的节点

本小节实现了一个简单的，会造成内存泄露的无锁栈。

```

template<typename>
class thread_safe_stack
{
public:
    struct node;

    void push(const T& data)
    {
        auto new_head = new node(data);
        new_head->_next = head.load();
        while (head.compare_exchange_weak(new_head->_next, new_head)) {}
    }

    shared_ptr<T> pop()
    {
        auto old_head = head.load();
        while (old_head && head.compare_exchange_weak(old_head, old_head->_next)) {} // old_head->_next might result in undefined behavior if it's
        deleted by other threads
        return old_head? old_head->_data: shared_ptr<T>();
    }

private:
    struct node
    {
        std::shared_ptr<T> _data;
        node* _next;

        node(T const& data): data(std::make_shared<T>(data)) {}
    };

    std::atomic<node*> head;
}

```

在此选择不释放节点（**造成内存泄露**）来解决当多个线程进行pop操作时, while--old_head->_next可能产生的未定义行为（如果old_head被别的线程删除了的话）。

7.2.2 Stopping those pesky leaks: managing memory in lock-free data

structures

如果只有一个线程进行pop操作，那么没有问题——因为push不会改变栈中已有的节点。

我们考虑多个线程进行pop操作，在此采取的策略为：

- 添加了_to_be_deleted链表用来维护待删除的（亦即已被pop出）的节点
- 当_thread_in_pop == 1时，删除节点

```

template<typename T>
class thread_safe_stack
{
public:
    struct node;

    void push(T const& data)
    {
        auto new_head = new node(data);
        new_head->next = head.load();
        while (!head.compare_exchange_weak(new_head->next, new_head)) {}
    }

    std::shared_ptr<T> pop()
    {
        ++ _thread_in_pop;
        auto old_head = head.load();
        while (old_head && ! head.compare_exchange_weak(old_head, old_head-
>next)) {}
        std::shared_ptr<T> res;
        if (old_head)
        {
            res.swap(old_head->data);
        }
        try_reclaim(old_head);
        return res;
    }

private:
    struct node
    {
        std::shared_ptr<T> _data;
        node* _next;
        node(T const& data): _data(std::make_shared<T>(data)) {}
    };

    void try_reclaim(node* old_head)
    {
        if (_thread_in_pop == 1)
        {
            auto nodes_to_delete = _to_be_deleted.exchange(nullptr);
            if (--_thread_in_pop == 0)
            {
                delete_nodes(nodes_to_delete);
            } else
            {
                chain_pending_nodes(nodes_to_delete);
            }
            delete old_head;
        } else
    }

```

```

        {
            chain_pending_node(old_head);
            --_thread_in_pop;
        }
    }

void chain_pending_node(node* n)
{
    chain_pending_nodes(n, n);
}

void chain_pending_nodes(node* nodes_to_delete)
{
    auto last = nodes_to_delete;
    for ( ; last->next; last = last->next) {}
    chain_pending_node(first, last);
}

void chain_pending_nodes(node* first, node* last)
{
    last->next = _to_be_deleted;
    while (!_to_be_deleted.compare_exchange_weak(last->next, first)) {}
}

void delete_nodes(node* nodes_to_delete)
{
    while (nodes_to_delete)
    {
        auto next = nodes_to_delete->next;
        delete nodes_to_delete;
        nodes_to_delete = next;
    }
}

std::atomic<unsigned> _thread_in_pop;
std::atomic<node*> _head;
std::atomic<node*> _to_be_deleted;
};

```

在此带来的问题时，当并发量高，亦即_thread_in_pop总是大于1时，无法删除节点，无法回收内存。

7.2.3 Deleting nodes that can't be reclaimed using hazard pointers

针对_thread_in_pop总是大于1的情况，本小节采用hazard pointer策略：

- 用_nodes_to_reclaim链表维护old_heads
- 每次pop操作都检查本线程的old_head是否有被别的线程使用（用定长数组hazard_pointers[max_hazard_pointers]存放每个线程使用着的old_head）
 - 若是，将old_head放入_nodes_to_reclaim

- 反之，删除old_head
- 无论成功与否，都遍历_nodes_to_reclaim，对于每个current节点，都通过outstanding_hazard_pointers_for查看hazrd_pointers数组中是否有引用current的线程，若无，则可以删除current

缺点：原子操作通常比非原子操作慢100倍，这样使得pop操作的开销很大。

7.2.4 Deleting nodes in use with reference counting

通过引用计数，实现了pop操作中的节点删除。